Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem

Darius Buntinas, Guillaume Mercier, William Gropp

Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem ⋆

Darius Buntinas [a,*], Guillaume Mercier [b], William Gropp [a]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*

[b] *LaBRI, Université Bordeaux I – INRIA Futurs*

Abstract

This paper presents the implementation of MPICH2 over the Nemesis communication subsystem and the evaluation of its shared-memory performance. We describe design issues as well as some of the optimization techniques we employed. We conducted a performance evaluation over shared memory using microbenchmarks. The evaluation shows that MPICH2 Nemesis has very low communication overhead, making it suitable for smaller-grained applications.

Key words: Parallel computing; Message passing; MPI; MPI implementation; Communication subsystem; Shared-memory

1 Introduction

The Message Passing Interface (MPI) standard has been designed to enhance portability in parallel applications, as well as to bridge the gap between the performance offered by a parallel architecture and the actual performance

⋆ This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

* Corresponding author.
Email addresses: buntinas@mcs.anl.gov (Darius Buntinas), mercierg@mcs.anl.gov (Guillaume Mercier), gropp@mcs.anl.gov (William Gropp).

delivered to the application. The level of achievable performance depends, however, on the implementation. Two critical areas determine the overall performance level of an MPI implementation. The first area is the low-level communication layer that the upper layers of an MPI implementation can use as foundations. The second area covers the communication progress and management. We designed an efficient communication subsystem, called Nemesis, that features very low overhead and is therefore suitable to serve as a basis for the MPICH2 software [1], an open source implementation of MPI.

The design and implementation of the Nemesis communication subsystem have been presented in [2]. In this paper, we describe how we ported MPICH2 over Nemesis and show the performance benefits of MPICH2 Nemesis. We also explain the improvements that have been made in the MPICH2 communication progress engine. The resulting MPICH2 software stack yields a very low latency and high bandwidth and compares favorably with competing software. The implementation also allows us to better assess both the overhead and the performance of MPI.

Section 2 gives an overview of the Nemesis communication subsystem. Section 3 describes how this communication subsystem has been integrated in MPICH2 as a new CH3 channel. We detail how we implemented several important features of the MPI2 standard. The various optimizations that MPICH2 gained are also explained. Section 4 presents a performance evaluation using shared-memory communication; in particular, we compare our implementation with the MPICH2 shm channel and other MPI implementations. Section 5 presents related work. Section 6 concludes this paper and discusses future work.

2 Overview of the Nemesis Communication Subsystem

In this section, we briefly describe the Nemesis communication subsystem. See [2] for a complete description of the design and implementation.

The Nemesis communication subsystem was designed to be a scalable, high-performance, shared-memory, multinetwork communication subsystem for MPICH2. The goals for our design, in order of priority, were scalability, high-performance intranode communication, high-performance internode communication, and multinetwork internode communication. The implication of ranking the goals this way is that we strive to minimize the overhead for intranode communication, even if this comes at some penalty for internode communication.

To achieve the goals of high scalability and low intranode overhead, we designed Nemesis using lock-free queues in shared memory. Thus, each process

needs only one receive queue, onto which other processes on the same node can enqueue messages without the overhead of acquiring a lock. Other designs would be to use a pair of receive queues per pair of processes or to use a single queue with a lock. On a large SMP, neither would be scalable, because of the $\mathcal{O}(N^2)$ number of queues needed or the contention on the lock, nor would they be efficient, because of the overhead of polling multiple queues or the overhead of acquiring and releasing a lock.

For internode communication, when a message is received from the network, a polling function for that *network module* enqueues the message onto the process's receive queue. A network module has a send queue onto which messages to be sent are enqueued. The send queue is analogous to a process's lock-free receive queue in that, when a process sends a message, it will enqueue the message onto the appropriate queue, whether it is a queue for another process on the same node or a send queue for a network module. This strategy simplifies the critical path when sending a message: No special action is taken when sending a message to a process on a remote node versus a process on the local node. Multiple networks can be supported by implementing additional network modules. Our current implementation supports internode communication over sockets, Myricom's GM and MX message-passing systems [3], and Quadrics [4].

After analyzing our initial design, we applied several optimizations. To reduce latency, we optimized the placement of the receive queue *head* and *tail* pointers and added a *shadow head* pointer to reduce L2 cache misses. We also gathered in the same cache line, variables that are often used together, to reduce the number of L1 cache misses in the critical path. For small SMP nodes, we used a *fastbox* mechanism to bypass the queues. A pair of buffers is allocated between each pair of processes. When sending a message, a process can bypass the queue by copying the message into the fastbox, if it is free, and setting a flag indicating a message is waiting. The receiving process then copies the message out of the fastbox and resets the flag. If the fastbox is full when a process is sending a message, the regular queue mechanism is used. This mechanism would not scale well for large SMPs and is used only for SMPs with a small number of processors. To improve bandwidth, we implemented architecture-specific memory copy functions. For ia32 and x86_64 architectures the memory copy function uses nontemporal store operations that bypass the cache. More details on these optimizations can be found in [2].

3 Integration into MPICH2

The communication portion of MPICH2 is implemented in several layers, as shown in Figure 1, and provides two ways to port MPICH2 to a communication subsystem. The ADI3 layer presents the MPI interface to the application layer

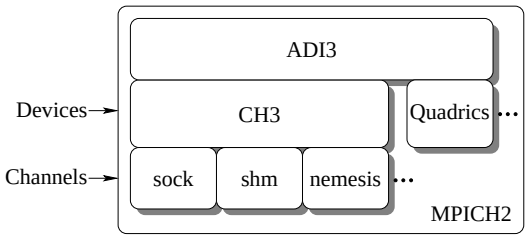


Fig. 1. Software layers of MPICH2

above it, and the ADI3 interface to the *device* layer below it. MPICH2 can be ported to a new communication subsystem by implementing an ADI3 device.

The figure shows the device for the Quadrics network. The figure also shows the CH3 device. The CH3 device presents the CH3 interface to the layer below it and provides another way for MPICH2 to be ported to a new communication subsystem: by implementing a *channel*. This interface has fewer functions than the ADI3 interface, making it significantly simpler to implement. Because the interface is simpler, however, it may not be able to take advantage of certain features provided by the communication subsystem, such as RDMA or collective operations.

We chose to port MPICH2 over Nemesis by implementing a CH3 channel. While our intent is to eventually implement an ADI3 device for Nemesis, implementing a CH3 channel allowed us to rapidly create a prototype and evaluate the implementation of Nemesis. We did, however, modify the CH3 layer in order to allow certain optimizations of the Nemesis channel. In the rest of this section we describe the basic design of the Nemesis channel and key optimizations.

3.1 Basic Design of the Nemesis Channel

To send a message, the CH3 layer calls a send function implemented by the channel, passing in a pointer to the message header a description of the data to be sent and a pointer to an MPI *request* object. The description of the data consists of an array of pointers and lengths (i.e., an *IOV*) that can be used to describe noncontiguous data. The Nemesis channel copies the header and data into a Nemesis receive queue element, called a *cell*, and fills in a short Nemesis header, then enqueues it on the appropriate receive queue or fastbox or sends it over the network to the appropriate remote node. If the CH3 message is larger than a cell, multiple cells can be used, since the cells are delivered in FIFO order.

If enough free cells are not available to send an entire message, the IOV describing the unsent data is saved in the request, which is then enqueued onto a pending-send queue. When free cells are available, the messages on the

pending-send queue are sent out. When all the data described by the IOV has been sent, the channel makes an up-call to CH3 to see whether there is more data to be sent. If there is, the IOV is reloaded; otherwise the request is marked as complete.

To receive a message, the Nemesis channel polls the receive queue and fastboxes. In order to reduce the overhead of unnecessarily polling too many fastboxes, the Nemesis channel polls only *active* fastboxes, which are the fastboxes of processes for which this process has posted a receive. Because fastboxes introduce a second path for messages between two processes, sequence numbers are used to maintain the order of messages.

When a cell is found, either in the receive queue or the fastbox, and there are no pending receives for that source process, the channel makes an up-call to CH3 with a pointer to the message header. If there is data to receive, CH3 will return an IOV along with a pointer to a request. The channel then copies the data from the cell to the user buffer described by the IOV. If the IOV describes more data than is contained in the cell, the IOV for the unreceived data is saved in the request, and the request is saved as a pending-receive corresponding to the process that sent the message. When the next cell from that process is received, the channel gets the saved request, and the new data is copied from the cell to the user buffer described by the IOV in the request. When all of the data described by the IOV has been received, the channel makes an up-call to CH3 to either reload the IOV, if there is more data to receive, or to mark the request as complete.

Because cells are allocated in shared memory, they are a limited resource. Hence, it is important to process a cell and copy out its data as soon as possible, so that it can be freed. This means that an unexpected message should be copied out of its cells and into a temporary buffer, as opposed to leaving the data in the cells until the receive has been posted. Unexpected messages are handled by the CH3 layer in just this way. If an unexpected message is received, CH3 creates a new request and passes back an IOV pointing to a newly allocated temporary buffer. So, the channel takes the same action whether the received message is unexpected or not. The message is copied out of the temporary buffer into the user buffer once a receive matching the message has been posted.

3.2 Large Message Transfer Using Rendezvous

While the shared-memory queue is very efficient for transferring small- to medium-sized messages, transferring large amounts of data through the queue may not be the most efficient method. High-performance networks have RDMA capabilities where data can be transferred directly from the user's source buffer

on one node to the user's destination buffer on another node, avoiding the data copies associated with using the queue. Some shared-memory machines, such as the SGI Altix, have similar mechanisms for processes on the same node. Even without such special mechanisms, using a queue still may not be the most efficient method of transferring large amounts of data between processes on the same node [5].

To support various mechanisms for transferring large messages, we defined the *Large Message Transfer* (LMT) interface and added it to CH3. Avoiding the queue can not only improve the bandwidth of the transfer but also reduce the impact on the application's data in the cache [5].

CH3 uses a rendezvous protocol when sending large messages, which ensures that a matching receive has been posted before the message data is sent. The rendezvous protocol is used primarily to avoid having to buffer the message if a matching receive has not been posted. The LMT interface is used together with the rendezvous protocol and allows the channel to piggyback information on the CH3 rendezvous messages. The channel implements three LMT functions for sending a message — `lmt_pre_send()`, `lmt_start_send()`, and `lmt_post_send()` — and three corresponding `_recv` functions for receiving a message.

Before sending a rendezvous RTS message, the sender calls `lmt_pre_send()`, specifying the destination and describing the data to be sent. This allows the channel to perform any required set up for the transfer, such as registering the source buffer. The channel returns a *send cookie*, which is variable-length data to be sent along with the RTS message to the receiver. Depending on the implementation, the cookie may contain memory registration keys, a description of the send buffer, and so forth.

Upon receiving the RTS and matching it with a posted receive, CH3 calls `lmt_pre_recv()`, passing in the send cookie and describing the receive buffer. This function allows the channel at the receiver to prepare for the transfer. The channel returns a *receive cookie* and specifies whether CH3 should send a rendezvous CTS message. Some data transfer mechanisms, such as when using an RDMA get operation, can start transferring data at this point and do not require action on the sender's side. In such a case a CTS message will not be sent. If required, CH3 will send the CTS message with the receive cookie. Next, CH3 calls `lmt_start_recv()`. At this point the channel at the receiver has all of the information it requires to start transferring data.

If a CTS message is sent by the receiver, then when it is received by the sender, CH3 calls `lmt_start_send()`, passing in the receive cookie from the CTS message. Now the sender has all of the information it requires to start transferring data.

The data is now ready to be transferred. Depending on the transfer mechanism, one side or both sides will participate in transferring the data. If both sides participate in the transfer, such as when copying through a buffer shared-memory on an SMP machine, then when the transfer is complete, the channel at the sender and receiver will call `lmt_post_send()` and `lmt_post_recv()`, respectively, to signal that the request has completed, and will perform post-transfer cleanup, such as deregistering memory or detaching from a shared-memory region. If, however, only one side is performing the transfer, such as when using an RDMA put or get, the channel on the side performing the transfer will send a DONE message to the other side. Upon receiving a DONE message, CH3 will call `lmt_post_send()` or `lmt_post_recv()`, as appropriate.

Some mechanisms may need to exchange additional information during the transfer. For example, the size of the data to be transferred may exceed the amount of memory that a network can register, so the message would need to be registered and transferred in sections. To handle such a case, the channel will send a COOKIE message to the remote side. Upon receiving a COOKIE message, CH3 calls the `lmt_handle_cookie()` function that the channel implements. The COOKIE message can be used to notify the remote side that additional memory has been registered or that part of the message has been successfully transferred so the memory can be deregistered.

Our current implementation uses the LMT interface for shared-memory communication and communication over the GM network. Large messages are transferred in shared-memory communication by copying through a shared buffer. The receiver allocates a shared-memory region and passes the information about the region to the sender in a receive cookie. On receiving the CTS with the receive cookie, the sender attaches to the shared-memory region and starts copying the data into the buffer. The data is copied by using a double buffer technique to allow both processors to perform the data copy operations simultaneously. Because both sides actively participate in the transfer, both sides know when the transfer has completed, so there is no need to send a DONE message.

When using GM to transfer large messages, we use RDMA put operations. In the `lmt_pre_` functions, the buffers are registered. Also, because the sender side needs to specify the target buffer for the message in the receiver's memory, the receiver includes that information in the receive cookie. In order to handle the case where the receive buffer is noncontiguous, the receiver sends the *dataloop* representation [6] of the MPI datatype to the sender in the receive cookie. The sender then uses one or more RDMA put operations to transfer the data. For very sparse datatypes it would be more efficient to transfer the data by packing it into an intermediate buffer and transferring it in a contiguous chunk rather than performing many small RDMA put operations. Our current implementation does not check for this case. We leave this for future work.

3.3 Bypassing the Posted Receive Queue

We performed another optimization to improve the latency of small messages by bypassing the CH3 posted receive queue in certain cases. In the current implementation of CH3 when a receive is posted by the application, CH3 first searches the unexpected message queue to see whether it has already received a matching message. If a matching message is not found, the request is posted on the posted receive queue. CH3 then calls the progress engine to check for incoming messages. When a new message is received, CH3 looks for a matching receive request by searching the posted receive queue and enqueues the message in the unexpected queue if the message is not found.

Notice that if a receive is posted for which there is no matching message in the unexpected message queue, and the matching message is waiting to be received on the Nemesis receive queue or network, the receive request is queued on the posted receive queue, only to be matched and dequeued in the next step when the progress engine is called and the matching message received. Our optimization implements a new function to call the progress engine with a receive request. As messages are received from the Nemesis receive queue, they are checked for a match with the receive request. Only when no matching messages are found on the receive queue is the request enqueued onto the posted receive queue. Note that if there already is a request on the posted receive queue that can possibly match the same message as the new receive request, we cannot use the optimization and, instead, add the new request to the receive queue as in the original implementation. This optimization reduced latency by about 18%, or 62 ns.

3.4 Shared-Memory Barrier Synchronization

The default implementation of barrier synchronization (i.e., `MPI_Barrier()`) in MPICH2 is implemented by using point-to-point messages. Furthermore, the algorithm does not differentiate between processes on the same node and processes on remote nodes. We optimized the barrier operation by using shared-memory variables to synchronize processes on the same node, and point-to-point messages to synchronize the processes on different nodes

The algorithm we use to perform a barrier by using shared memory has been previously presented in [7]. In this algorithm, processes atomically increment a `count` variable in shared memory, and then wait for a `signal` variable, also in shared-memory, to change. The last process to increment `count`, resets `count` for the next barrier and then flips the value of `signal` variable. To reduce the number of cache invalidations, we located each shared-memory variable in its own cache line. This way, while the processes who have already entered the barrier are polling on the `signal` variable, cache misses won't be

```
1       barrier() {
2           int prev, sense;
3
4           sense = sig;
5           prev = FETCH_AND_INC(counter);
6           if (prev == P - 1) {
7               count = 0;
8               signal = 1 - sense;
9           } else
10               while (signal == sense)
11                  SKIP;
12      }
```

Fig. 2. Shared-memory barrier algorithm. `P` is the number of processes participating in the barrier, and `signal` and `count` are variables in shared memory.

generated when other processes increment the `count` variable. Only when the last process flips the value of the `signal` variable will all of the cache lines for the waiting processes be invalidated and have to be reloaded. Figure 2 shows the pseudocode for this algorithm.

In order to handle the case where not all processes participating in the barrier are on the same node, each process identifies the set of processes running on its node. The process with the lowest global rank on each node is selected as the leader. The leader processes then identify the set of all leader processes on all nodes. An additional shared variable, `signal_leader`, is used in these types of barriers. To perform the barrier, the nonleader processes perform an atomic increment on `count` and then wait on `signal` to change. The last nonleader process that increments `count` sets `signal_leader` to true before waiting on the `signal` variable. The leader process waits for `signal_leader` to become true and then performs a message-based barrier with the other leader processes, using the dissemination barrier algorithm [8]. Once the leader process completes the message-based barrier, it resets the `count` and `signal_leader` variables and then flips the value of the `signal` variable, allowing the other processes to exit the barrier.

One issue that we had to address is the allocation of the shared-memory variables. Because in Nemesis shared memory is allocated once at initialization time and cannot be expanded, shared memory is a limited resource. For this reason we cannot simply allocate a set of shared-memory variables for every MPI communicator created. Instead, we allocate the variables to a communicator when the first barrier is performed on that communicator. This allocation is done by organizing the variables in a list, keyed on the communicator's context ID, which is initially set to a NULL value. The leader process of a node walks this list performing an atomic compare-and-swap comparing the key of

each element with the NULL value. When the compare-and-swap succeeds, the leader process has successfully allocated the element. At this point because only the leader process knows which list element has been allocated, the processes cannot use shared memory to perform the barrier. Instead, the processes perform a message-based dissemination barrier. In the messages being exchanged, the index of the list element is disseminated among the processes. After this barrier, the index of the list element is known by all processes on the node, so subsequent barriers can use shared memory. The list element is freed when the communicator is destroyed.

If no free list elements are available, the leader process disseminates the NULL value in the message-based barrier to the other processes on the node, and the processes will again attempt to allocate a list element in the next barrier. We intend to modify Nemesis to be able to dynamically add shared memory after initialization time. This feature would eliminate the possibility of running out of list elements.

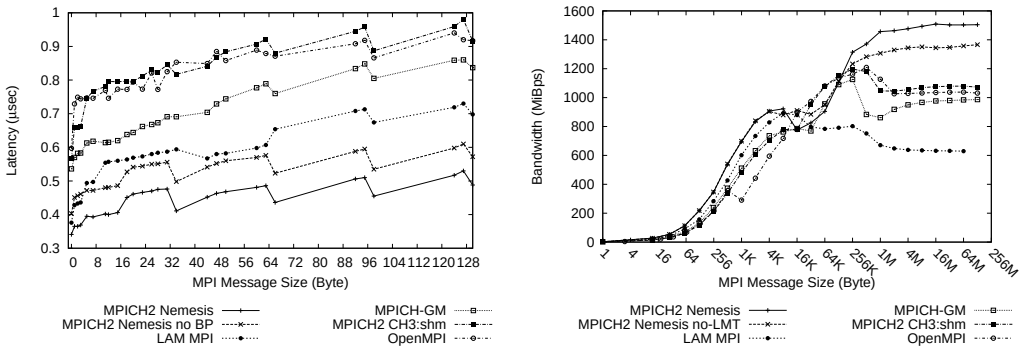
4 Performance Evaluation of MPICH2 over Nemesis

In this section we evaluate the shared-memory performance of our implementation of MPICH2 over the Nemesis communication subsystem. We present a microbenchmark evaluation on a 2 GHz dual-processor dual-core Opteron 280 machine with 2 GiB of memory. An evaluation of the optimized barrier implementation was performed on a cluster of eight dual-processor dual-core Opteron 275 machines with 4 GiB of memory. We configured MPICH2 with the `--enable-fast` option that disables error checking and configured Open-MPI to disable error checking and support for heterogeneous clusters, which should improve the performance for those implementations. All implementations were compiled by using the `-O3` optimization.

4.1 Latency and Bandwidth

We compare our implementation to LAM/MPI [9] version 7.1.2, OpenMPI [10] version 1.1, MPICH-GM [3] version 1.2.6..14b, and MPICH2 version 1.0.3 configured with the CH3 *shm* channel that communicates by using shared memory. All these MPI implementation use shared-memory intranode communication. Except where noted, the results for MPICH2 Nemesis have both the LMT and posted receive queue bypass optimizations applied. We measured latency and bandwidth using Netpipe [11]. Figure 3 shows these results.

The latency graph in Figure 3(a) shows two data series for MPICH2 Nemesis. The results shown by the data series labeled "MPICH2 Nemesis no BP" were taken without the posted receive queue bypass optimization. This optimization



(a) Latency (b) Bandwidth

Fig. 3. Shared-memory performance of MPI implementations

improves latency by about 62 ns, resulting in a zero-byte latency of 341 ns. With the optimizations applied, MPICH2 Nemesis has lower latency than the other MPI implementations. Even up to 128 bytes, the MPICH2 Nemesis latency is just over 500 ns.

Figure 3(b) shows the bandwidth comparison. Nemesis uses an optimized memory copy routine that uses nontemporal store operations. Using the nontemporal copy routine results in dramatically higher bandwidth for MPICH2 Nemesis compared to the other MPI implementations. The results shown by the data series labeled "MPICH2 Nemesis no-LMT" were taken without applying the LMT optimization to MPICH2 Nemesis. The LMT optimization improves bandwidth by about 130 MiBps for large messages, resulting in a peak bandwidth of over 1,500 MiBps. Notice that for MPICH2 Nemesis, at 16 KiB the bandwidth of the non-LMT implementation is a little higher than the implementation with LMT. The reason is that at 16 KiB, the communication protocol switches from eager to rendezvous and additional setup is performed for LMT. The figure shows that MPICH2 Nemesis has higher bandwidth than the other MPI implementations except for messages between about 16 KiB and 256 KiB. We intend to perform additional tuning to improve the medium-sized message bandwidth and find the optimal message size for the crossover from eager to rendezvous protocol.

4.2 Barrier

We evaluated our shared-memory barrier optimization by comparing the latency of the barrier operation using the optimization to the default implementation. The benchmark consists of measuring the average time to complete 100,000 barriers. We ran this benchmark on a cluster of eight machines with four processors each. The machines were connected with a gigabit Ethernet network. Figure 4 shows the results of this benchmark. Notice that the optimized implementation has much lower latency than that of the default

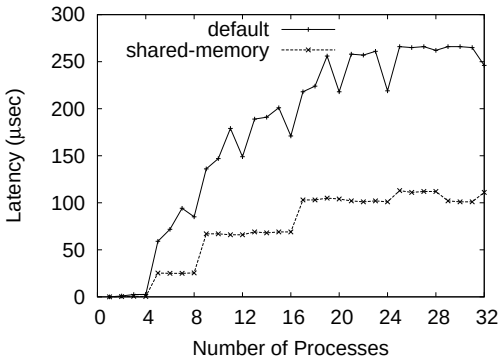


Fig. 4. Barrier latency for the default MPICH2 implementation and optimized shared-memory barrier implementation

MPICH2 implementation. For a four-process barrier on a single node, the latency of our optimized implementation is 0.4 μs, whereas the latency for the default implementation is 2.6 μs. This is more than a factor of 6 improvement. For barriers across processes on different nodes, the optimized implementation shows regular jumps in performance corresponding to the $\mathcal{O}(\lg n)$ number of messages required to perform the dissemination barrier between the leader processes on each node. The default implementation also shows an $\mathcal{O}(\lg n)$ increase in latency; however, whereas the latency of the optimized implementation increases as a function of the number of nodes, the latency of the default implementation increases as function of the number of processes. Furthermore, because more than one process on one node may be sending messages to processes on another node, contention on the network interface increases the average latency of those messages. For multinode barriers, the optimized implementation shows a factor of improvement greater than 2 over the default implementation.

4.3 Instruction Count

One of the goals of our implementation is to streamline the critical path. One way of measuring our success is by counting the number of instructions required to send or receive a message. Using the PAPI [12] performance counter interface, we measured the instruction count for send and receive eight-byte messages. When measuring the instruction count for the receive operations, we wanted to avoid counting instructions performed polling while waiting for the message to arrive because the waiting time can vary quite a bit. To do this we added a delay equal to the round trip time before starting to count instructions and performing the receive. This ensured that the incoming message had arrived and was waiting at the receive queue when `MPI_Recv` was called. Table 1 shows these results. All MPI implementations were compiled with the -O3 optimization level, except for MPICH-GM, where the unoptimized code had fewer instructions.

Table 1
Instruction count for sending and receiving an eight-byte message.

MPI Implementation	`MPI_Send`	`MPI_Recv`	Total
OpenMPI	550	1,745	2,295
MPICH-GM	455	617	1,072
LAM MPI	436	472	908
MPICH2 CH3:shm	311	748	1,059
MPICH2 Nemesis no BP	241	712	952
MPICH2 Nemesis	**241**	**259**	**500**

The row labeled "MPICH2 Nemesis no BP" shows the instruction counts when the posted receive queue bypass optimization was not applied. The results show that this optimization reduces the combined send and receive instruction count by almost half. With the optimization, the combined instruction count for MPICH2 Nemesis is less than 22% that of OpenMPI, less than 50% that of MPICH2 CH3:shm and MPICH-GM, and 55% that of LAM MPI. The instruction counts show that the critical path in our implementation is already quite efficient; however, we believe that we still can further streamline the critical path and improve cache utilization, which will reduce overall latency for small messages.

4.4 The Halo Benchmark

One of the benchmarks we used to predict the application performance of MPICH2 Nemesis was the Halo benchmark [13]. This benchmark simulates a nearest neighbor exchange of a 1 to 2 row and column "halo" from a 2D array. The authors of the Halo benchmark state that performance of the Halo benchmark correlates well with the performance of their layered ocean model application. We ran the benchmark on the Opteron machine using four processes.

The Halo benchmark performs the halo exchanges by using several different algorithms. Figure 5 shows the results for the algorithm that performed best for each MPI implementation. The algorithm that used `MPI_SendRecv()` performed best in MPICH2 Nemesis, MPICH-GM, and OpenMPI. In MPICH2 CH3:shm, the algorithm using `MPI_ISend()` and `MPI_IRecv()` performed best. In LAM MPI, the best performance was seen when using the algorithm that used persistent sends and receives, where the receives are posted before the send operations are called. In the figure, we see that MPICH2 Nemesis performs considerably better than the other implementations for all tile sizes. Of the others, MPICH2 CH3:shm performs better than LAM MPI, MPICH-GM, and OpenMPI for small tile sizes. For larger tile sizes MPICH-GM performs better than MPICH2 CH3:shm, LAM MPI, and OpenMPI. The performance of this benchmark is dominated by latency for small tile sizes and by band-

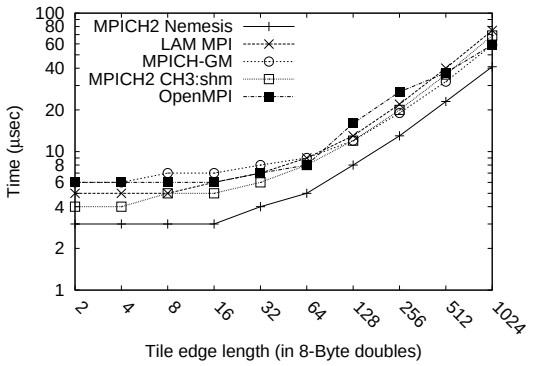


Fig. 5. Results of the Halo benchmark using four processes

width for large tile sizes. The factor of improvement for MPICH2 Nemesis over the other implementations ranges from 1.5 to 2.6. This suggests that MPICH2 Nemesis should perform well on applications that are sensitive to latency or need high bandwidth.

5 Related Work

Other high-performance MPI implementations also use shared-memory to improve intranode communication. In order to manage concurrent access to queues in the shared-memory region, some implementations, e.g., MPICH-PM [14], use spin-locks. By using spin-locks the overhead of accessing OS-locks is eliminated. However, because access to queues in shared memory is serialized, this can impact performance in cases where messages are rapidly received by one process while other processes are rapidly sending to that process.

To address such potential serialization, an implementation can use lock-free data structures. Lock-free data structures are used in MPI-BIP [15,16], AM-II [17,18], and Nemesis. MPI-BIP uses separate queues for each pair of processes communicating via shared-memory. This method may waste shared-memory resources when some pairs of processes never communicate. AM-II and Nemesis implement lock-free queues using atomic operations, such as swap and compare-and-swap. By using atomic operations, a lock-free queue can be implemented that allows multiple processes to access the queue concurrently, so each process needs only a single queue to be able to receive messages from any other process. In AM-II, to send a message, a process will attempt to dequeue a packet from a free queue, but if another process dequeued the same packet before the first process finished, the first process must try again. This can lead to starvation and considerable traffic in the memory system when many processes are trying to send to the same processes at the same time. Nemesis uses a lock-free queue which does not require repeated attempts, thus ensuring fairness and reduces memory bus traffic. Details on the implementation of lock-free queues in Nemesis can be found in [2].

In order to improve intranode throughput, MPI-BIP and MPICH-PM use kernel modules to copy data directly from the source process's buffer space to the destination process's buffer. The data can then be transferred with a single copy, as opposed to two copies if the data were copied through a shared-memory segment. Nemesis does not provide such a single-copy data copy mechanism itself, but the LMT interface is general enough that support for such a mechanism could be easily added where such a mechanism exists.

6 Discussion and Future Work

In this paper we have presented our new implementation of MPICH2 over the Nemesis communication subsystem. We evaluated the shared-memory communication and barrier performance of our implementation on 4-core Opteron machines using microbenchmarks. Our implementation achieved a zero-byte latency of 341 ns and a 128-byte latency of just over 500 ns. The peak bandwidth of our implementation was over 1,500 MiBps. Our optimized barrier implementation achieved a factor of improvement greater than 2 over the default implementation. We also measured the number of instructions required to send and receive MPI messages. MPICH2 Nemesis uses only 500 instructions to send and receive an eight-byte messages. To evaluate application-level performance, we used the Halo benchmark, which favors low-latency and high-bandwidth MPI implementations, and saw a factor of improvement from 1.5 to 2.6 compared to the other implementations we evaluated. These results show that MPICH2 Nemesis has an efficient implementation of shared-memory communication, which achieves low latency and high bandwidth. Moreover, the results indicate that MPICH2 Nemesis would be especially suitable for smaller-grained applications, which are sensitive to latency and bandwidth.

Future work on MPICH2 Nemesis is to implement Nemesis as a full ADI3 device, which should further improve performance. We also intend to implement other optimized collective communication operations that take advantage of shared memory, as well as collective operation primitives provided by network interfaces.

References

[1] Argonne National Laboratory, MPICH2, `http://www.mcs.anl.gov/mpi/mpich2`.

[2] D. Buntinas, G. Mercier, W. Gropp, Design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem, in: S. J. Turner, B. S. Lee, W. Cai (Eds.), Proceedings of the 6th IEEE International

Symposium on Cluster Computing and the Grid (CCGRID '06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 521–530.

[3] Myricom, `http://www.myri.com`.

[4] Quadrics Supercomputers World Ltd., QsNet high performance interconnect, `http://www.quadrics.com/`.

[5] D. Buntinas, G. Mercier, W. Gropp, Data transfers between processes in an SMP system: Performance study and application to MPI, in: Proceedings of the 2006 International Conference on Parallel Processing (ICPP 2006), IEEE Computer Society, Washington, DC, USA, 2006, pp. 487–496.

[6] R. Ross, N. Miller, W. D. Gropp, Implementing fast and reusable datatype processing, in: J. Dongarra, D. Laforenza, S. Orlando (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, no. LNCS2840 in Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 404–413.

[7] J. Mellor-Crummey, M. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computer Systems 9 (1) (1991) 21–65.

[8] D. Hensgen, R. Finkel, U. Manber, Two algorithms for barrier synchronization, International Journal of Parallel Programming 17 (1) (1988) 1–17.

[9] G. Burns, R. Daoud, J. Vaigl, LAM: An open cluster environment for MPI, in: Proceedings of Supercomputing Symposium, 1994, pp. 379–386.

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.

[11] Q. O. Snell, A. R. Mikler, J. L. Gustafson, Netpipe: A network protocol independent performace evaluator, in: Proceedings of Internation Conference on Intelligent InformationManagement and Systems, 1996.

[12] S. Browne, C. Deane, G. Ho, P. Mucci, PAPI: A portable interface to hardware performance counters, in: Proceedings of Department of Defense HPCMP Users Group Conference, Monterey, California, 1999.

[13] A. J. Wallcraft, The Halo benchmark, `http://www.sdsc.edu/SciComp/PAA/Benchmarks/Portal/Halo/halo.html` (May 1998).

[14] T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, P. H. Beckman, Implementation and evaluation of MPI on an SMP cluster, in: Parallel and Distributed Processing – IPPS/SPDP'99 Workshops, Vol. 1586 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 1178–1192.

[15] P. Geoffray, L. Prylli, B. Tourancheau, BIP-SMP : High performance message passing over a cluster of commodity SMPs, in: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, 1999, p. 20.

[16] L. Prylli, B. Tourancheau, R. Westrelin, The design for a high-performance MPI implementation on the Myrinet network, in: Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, London, UK, 1999, pp. 223–230.

[17] S. Lumetta, D. Culler, Managing concurrent access for shared memory active messages, in: 12th International Parallel Processing Symposium, 1998, p. 272.

[18] S. S. Lumetta, A. M. Mainwaring, D. E. Culler, Multi-protocol active messages on a cluster of SMP's, in: Proceedings of Supercomputing '97, San Jose, California, 1997.